Calibration and Augmented Reality

Chirag Dhoka Jain, Keval Visaria
Project 4 Submission
course - ***EECE5330***
{dhokajain.c, visaria.k}@northeastern.edu
March 18, 2024
Northeastern University

I. Introduction

This project embarks on enhancing digital and physical world interactions through camera calibration and augmented reality (AR) techniques. Initially, it targets identifying and capturing the corners of specific patterns, like chessboards, within a video feed, critical for camera calibration. Calibration is pivotal, translating 3D real-world points into 2D camera image points by analyzing several target images from different perspectives. This process ensures accurate scene interpretation from any angle.

Once calibrated, the system can real-time track the camera's position relative to the target, essential for overlaying virtual objects onto the real world convincingly. These virtual elements maintain correct perspective and orientation as the camera moves, integrating seamlessly into the physical space. Moreover, the project explores robust feature detection within the environment, independent of specific patterns. This approach broadens the system's interactive capabilities, allowing for more sophisticated AR applications.

A. *Task 1: Detect and Extract Target Corners*

In this task, we've chosen to focus on using a checkerboard as our target for camera calibration. This choice is guided by the checkerboard's high contrast and distinct pattern, which significantly aids in the detection and extraction of precise corner points. Our system utilizes OpenCV functions—specifically,`'findChessboardCorners'`, `'cornerSubPix'`, and `'drawChessboardCorners'`—to identify these corners within the video feed, refine their positions for accuracy, and then visually represent them for verification. In our implementation, we use a vector,

```
std::vector<cv::Point2f> corner_set;
```

corner-set; to store the coordinates of detected corners, where each Point2f represents a corner's (x, y) position.
However, while using a checkerboard offers considerable advantages, it's not without its challenges:
Lighting Conditions: The system's ability to detect the checkerboard can be severely impacted by inadequate lighting, causing shadows or reflections that obscure the pattern.
Partial Visibility: If the entire checkerboard is not within the frame, the system might struggle to recognize the pattern and accurately locate corners.
Flat Surface Requirement: The checkerboard needs to be on a flat surface. Any bending or folding of the target can lead to inaccurate corner detection.
Consistent Pattern Size: The checkerboard pattern size must be known and consistent; variations can confuse the detection algorithm.
These challenges necessitate a controlled environment to some extent, ensuring that the checkerboard is fully visible, well-lit, and placed on a flat surface for the system to function optimally. Our initial task has been to create a reliable system that can overcome these hurdles, setting a solid foundation for the subsequent calibration process.

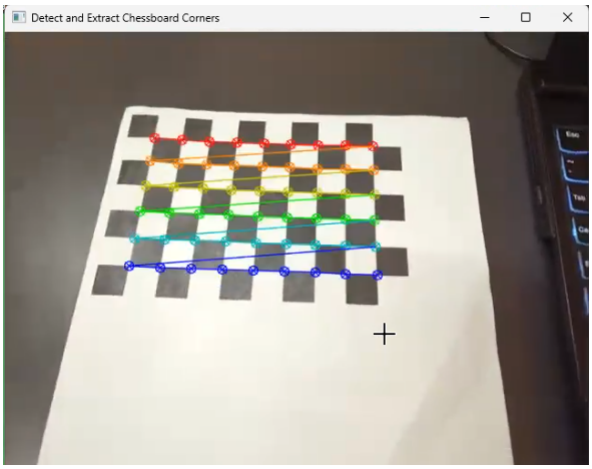

Fig. 1. Detection and extraction of chessboard corners

B. *Task 2: Select Calibration Images*

In our calibration process, we've introduced an interactive step where users can select specific images for calibration by pressing the 's' key. This action triggers the storage of the detected corners' locations in both 2D and 3D space. Here's a simple breakdown of how it works:

Press 's' Key: When viewing an image where the chessboard corners are correctly identified, pressing the 's' key saves these corner points.
2D Corner Points: The exact locations of the chessboard corners in the image (2D space) are stored in a list called $corner\_list$.

3D World Points: Simultaneously, a corresponding set of 3D coordinates representing the corners' positions in the real world, known as $point\_set$, is created. This set assumes a fixed distance between each corner, forming a grid that matches the chessboard's layout.
Recording Process: The system calculates and records the 3D coordinates based on the chessboard's configuration, with the origin at the upper left corner and the Z-axis pointing towards the viewer. This mapping remains constant, independent of the chessboard's orientation in the images.
Output: The console prints the coordinates of the corners for verification, signaling the completion of the recording phase with a "Finish Recording" message.
This method ensures that we have an accurate representation of the chessboard's geometry in both the image and real-world dimensions, crucial for the precise calibration of the camera system.

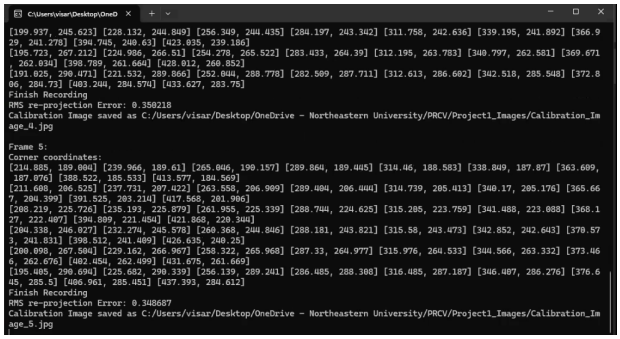

Fig. 2. Input image output

C. *Task 3: Calibrate the Camera*

In this task, we focus on calibrating the camera to enhance the accuracy of our augmented reality (AR) system. Calibration is vital for correcting any distortions and ensuring that our system can accurately overlay virtual objects in the real world.
Starting Calibration: When at least five images of the calibration target (like a chessboard) have been selected, you can initiate the calibration process by pressing the 'c' key.
Calibration Process: The system uses the previously stored corner points from the images and their corresponding 3D world points to adjust the camera's settings, minimizing errors between the expected and actual corner locations.
Adjusting Camera Settings: We specifically adjust the camera matrix and distortion coefficients. The camera matrix contains information like the focal length and the optical center of the camera, which are crucial for accurately projecting 3D points onto the 2D image.
Before and After: The system outputs the camera matrix and distortion coefficients both before and after calibration, allowing you to see the adjustments made.
Reprojection Error: Finally, the system calculates the reprojection error, which measures the accuracy of the calibration. A lower error means a more accurate system, ideally less than one pixel.

$$\text{Camera Matrix} = \begin{bmatrix} 1 & 0 & \frac{\text{frame.cols}}{2} \\ 0 & 1 & \frac{\text{frame.rows}}{2} \\ 0 & 0 & 1 \end{bmatrix}$$

This formula initializes the camera matrix with assumptions about pixel squareness and the optical center's location, setting the stage for fine-tuning during calibration.

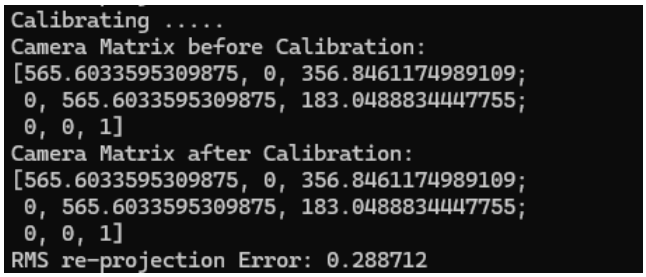

Fig. 3. Target Image

D. *Task 4: Calculate Current Position of the Camera*

In this task, we're focusing on determining the camera's current position relative to a detected target, like a chessboard, using the calibration parameters we previously calculated. This process involves analyzing each frame of the video feed to detect the target and then calculating the camera's position (rotation and translation) in relation to this target.

When you press the 'm' key, the program will start or stop displaying the rotation and translation matrices that represent the camera's position:
Rotation Matrix: Shows how the camera is oriented in space. As you move the camera from side to side, you'll notice changes in these values, reflecting the camera's rotation around the target.
Translation Matrix: Indicates the camera's location in relation to the target. Moving the camera closer or further away, as well as side to side, will change these numbers, showing the camera's movement through space. These matrices make sense when you consider the physical movements being performed. For instance:
Moving the camera to the left or right will primarily alter the values in the rotation matrix, as the camera's view angle changes in relation to the target.
Moving the camera closer or further away will adjust the translation matrix values, reflecting the change in distance between the camera and the target.
These values are crucial for understanding how the camera is positioned in 3D space, which is essential for creating accurate augmented reality overlays and for other applications that require precise knowledge of the camera's orientation and location.

E. *Task 5: Project Outside Corners or 3D Axes*

In this task, we're using the camera's position information, determined earlier, to project 3D axes onto the image plane,

creating a visual reference that moves in real-time with the camera or target. By pressing the 'a' key, the program draws these 3D axes originating from the chessboard's corner, extending into three dimensions along the board.

Press 'a' Key: Activates or deactivates the display of 3D axes on the screen.

Detection and Projection: The program detects the chessboard's corners and calculates the camera's orientation and position relative to it. It then projects virtual 3D axes from one corner of the chessboard into the image.

Visual Feedback: As you move the camera side to side:

The red line represents the X-axis.

The red line represents the X_{axis}.

The green line represents the Y_{axis}.

The blue line represents the Z_{blue}. (pointing outwards from the chessboard).

Real-time Changes: Moving the camera alters the axes' appearance on the screen, reflecting the camera's new position and orientation. This visual feedback helps in understanding how the camera's movement affects its perceived position in 3D space.

This dynamic projection of 3D axes onto the live video feed provides a tangible way to see how the camera's movements around the chessboard change its perspective and position relative to the chessboard. These projected axes are crucial for developing more complex AR applications, allowing developers to anchor virtual objects to the real world accurately.

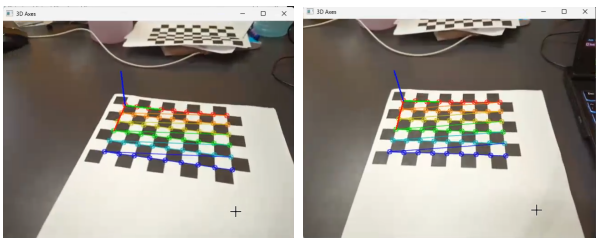

Fig. 4. Output images of 3D-Axis

F. Task 6: Classify new images

In this exciting task of our project, where the magic of augmented reality begins to take shape. Here, we delve into creating and embedding virtual 3D objects into live video feeds, making it appear as though these objects coexist within the real world. This process not only enhances the visual appeal but also paves the way for innovative augmented reality solutions.

1. Setting the Stage:

Calibration is Key: We start by calibrating our camera using `'calibrateCamera()'`, which fine-tunes the camera matrix and distortion coefficients. This step ensures that our virtual objects will appear correctly positioned and proportioned within the scene.

Capturing Camera Pose: Understanding our camera's exact stance in the world—its position and orientation—is crucial. We achieve this through `'solvePnP'`, resulting in rotation and translation matrices that tell us where our camera is looking from.

2. Crafting Virtuality:

Building Blocks: Virtual objects are crafted in 3D space, defined by vertices that sketch out their form—be it the sleek sides of a cylinder, the sharp edges of a pyramid, or the solid faces of a cube.

Versatile Design: Through a function like `'draw3dObject()'`, we breathe life into these shapes, setting parameters for their size and detailing their geometric structure.

3. Bringing Objects into View:

Transitioning to Camera View: The `'projectPoints()'` function is instrumental in translating the 3D coordinates of our virtual objects into the camera's perspective, effectively shifting them from their world space into our view.

Projection Mastery: These transformed vertices are then seamlessly projected onto the 2D plane of our current image frame, giving the illusion that they inhabit the physical space before the camera.

Artistry in Augmentation: To complete the illusion, cv::line() draws the outlines of these objects on the video feed, stitching the vertices together into recognizable shapes.

This intricate dance of calibration, projection, and visualization not only captivates the viewer but also demonstrates the profound potential of augmented reality

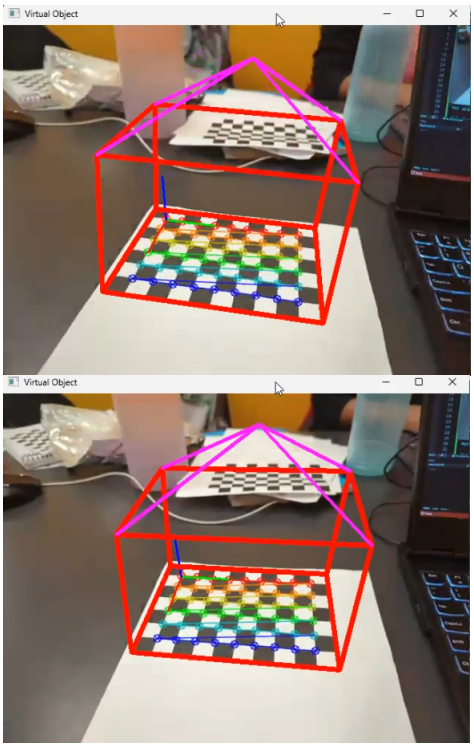

Fig. 5. 3D object into live frame

G. Task 7: Detect Robust Features

In this task, we delve into the realm of `'SURF'` (Speeded Up Robust Features) for robust feature detection, emphasizing its adaptability and precision in identifying key points within

an image. Like the Harris method's focus on intensity variations, `'SURF'` leverages the `'minHessian'` parameter to discern features that stand out due to their stability across varying conditions such as lighting changes and viewpoint shifts. This parameter acts as a filter, sifting through the image to pinpoint only the most distinctive features, akin to identifying the craggiest peaks in a mountain range, ensuring that the detected points are truly significant.

The essence of SURF, augmented by adjusting the `'minHessian'` value, mirrors the meticulous process of corner detection and non-maximum suppression found in Harris corner detection.

By setting a higher `'minHessian'` threshold, SURF becomes akin to a fine-tuned instrument, zeroing in on the most salient features, which are crucial for tasks requiring high fidelity, such as object recognition and augmented reality applications.

Increasing the minimum Hessian value tends to result in detecting fewer keypoints overall. While this can lead to faster processing times and potentially more robust features being retained, there is a risk of missing keypoints in areas of the image with lower contrast or texture. Consequently, setting a higher minimum Hessian value may lead to a reduction in the number of keypoints detected, which could impact the accuracy of subsequent tasks such as image matching or object recognition.

`'minHessian value = 10000'`

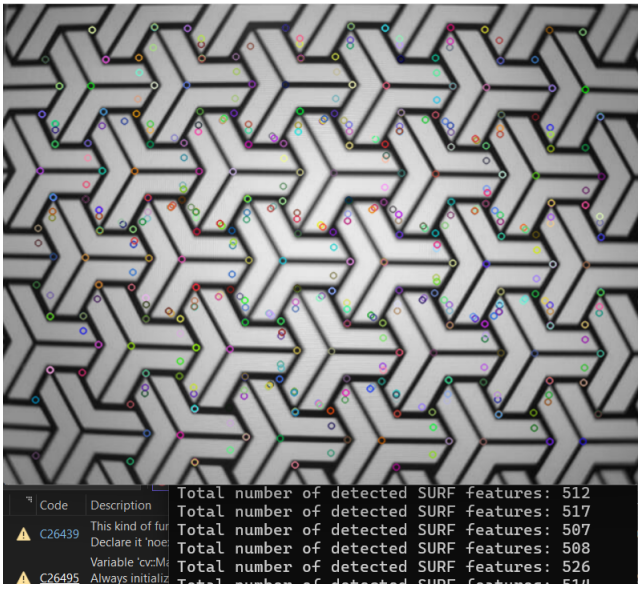

Fig. 6. Output of surf features detected on geometric patterns

Lowering the threshold, conversely, broadens the feature spectrum, capturing a wider array of points but possibly including less distinctive ones.

Lowering the minimum Hessian value tends to result in detecting more keypoints overall. This can be advantageous in scenarios where fine details or intricate patterns need to be captured, as it increases the likelihood of identifying relevant features. However, it may also lead to detecting more noise or spurious keypoints in regions with little meaningful structure. While a lower minimum Hessian value may enhance the sensitivity of feature detection, it can also increase computational overhead, as more keypoints need to be processed and analyzed. `'minHessian value = 2500'`

Fig. 7. Output of surf features detected on geometric patterns

This selective sensitivity, facilitated by the `'minHessian'` parameter, is instrumental in crafting a tailored feature detection strategy that underpins the creation of a rich, interactive AR experience, ensuring virtual elements integrate seamlessly

with the real world, anchored firmly to these reliable feature points.

`'minHessian value = 6000'`

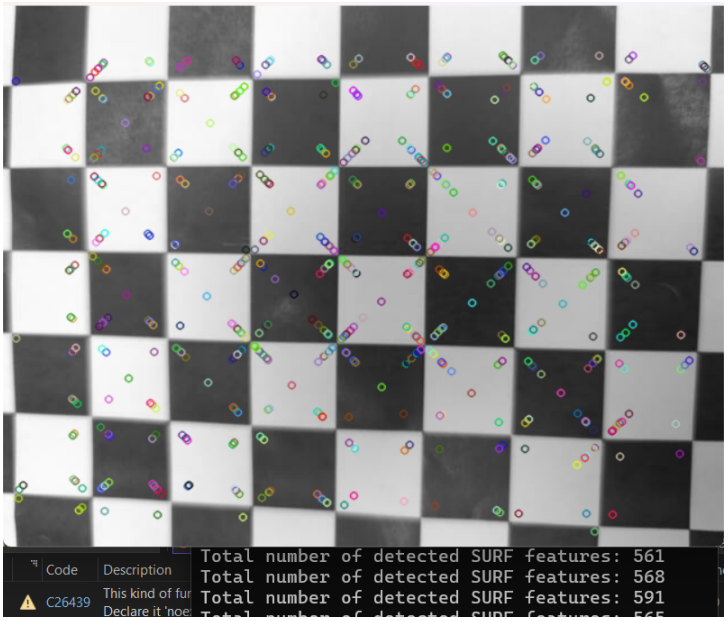

Fig. 8. Output of surf features detected on Checkerboard

H. *Task : Extensions*

1) *Get your system working with multiple targets in the scene*

In this task, we tackled the challenge of identifying multiple objects in an image, specifically focusing on detecting ArUco markers—a type of barcode that can be recognized by computers.

Setting Up the Detector: We started by setting up a tool specifically designed to look for these `'ArUco'` markers. Think of this like giving our computer a magnifying glass to find and recognize these special patterns.

Choosing a Marker Dictionary: `'ArUco'` markers come in various shapes and sizes, so we selected a specific set (or dictionary) of these markers called `'DICT_6X6_250'`. This is like choosing a book of patterns the computer is allowed to look for.

Looking for Markers: With our detector ready and the right book of patterns in hand, we then asked our computer to scan through the video frame by frame, searching for any markers from our selected set.

Identifying and Highlighting Markers: Whenever the computer found a marker, it made a note of where it was in the image and even drew a box around it to show us its location. In the end, we displayed the image with all the found markers highlighted, allowing us to see exactly where they were. By using this method, we efficiently taught our computer to spot these unique `'ArUco'` markers in any scene, paving the way for various applications, including augmented reality experiences, where objects in the real world can be enhanced or annotated digitally.

Fig. 9. Multi target with ARuco markers

2) *Test out several different cameras*

In this task, we did comparative study on camera inputs and their impact on RMS reprojection error, we analyzed data from three distinct sources: a standard laptop camera, a Google Pixel 7 phone camera, and a Logitech webcam. The findings were revealing, with the laptop camera recording a reprojection error of approximately 0.9, significantly higher than the errors from the Google Pixel 7 and Logitech webcam, which hovered around 0.3.

This disparity underscores the critical influence of camera quality on spatial measurement accuracy. Cameras with advanced optics and sensors, such as those in the Google Pixel 7 and high-end Logitech webcams, offer superior precision, essential for applications demanding exact spatial mapping. Conversely, the basic hardware of typical laptop cameras results in less accurate measurements, impacting the reliability of data for precise applications.

Our study highlights the necessity of careful camera selection in projects requiring meticulous spatial accuracy, such as augmented reality and 3D modeling, where the choice of camera technology can significantly affect the outcome's precision.

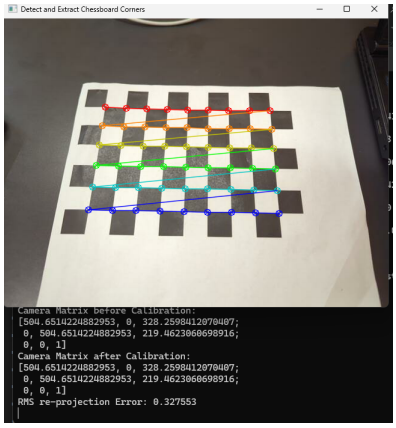

Fig. 10. Detection using phone camera and result

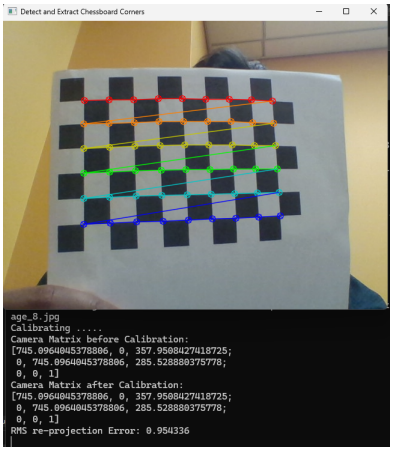

Fig. 11. Detection using laptop camera and result

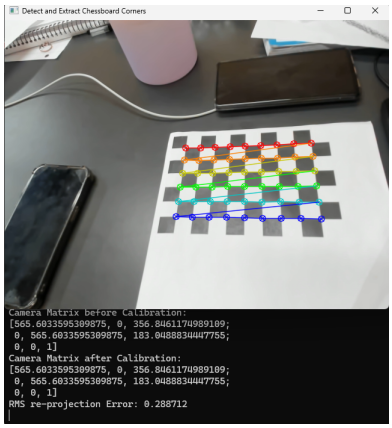

Fig. 12. Detection using logitech standby camera and result

3) *Enable your system to use static images with targets*

In this task, we aimed to overlay a static image (like a picture or a pattern) onto a specific target area within a live video frame, focusing on areas like a chessboard detected in the scene. Here's a simplified breakdown of how we achieved this:

1. Grayscale Conversion: First, the live video frame is converted to grayscale, simplifying further processing steps by reducing the complexity of dealing with color information.
2. Identifying Correspondence: We pinpoint corners of the static image that match with the corners of the chessboard in the video. This ensures that when we overlay the image, it aligns perfectly with the target area.
3. Perspective Transformation: The `'warpPerspective'` function is key here. It adjusts the static image to match the perspective of the chessboard in the video. This process involves stretching, shrinking, and rotating the image so it looks as if it's part of the original scene.
4. Masking for Isolation: A mask is created to isolate the chessboard area. This is done using the `'fillConvexPoly'` function, which draws a shape (in this case, around the chessboard) on a blank canvas to create a mask.
5. Inverse Masking: To overlay our image without losing the rest of the scene, we create an inverse mask using the `bitwise_not'` function. This mask removes the chessboard from the original frame, leaving a blank space where our image will go.
6. Blending the Image: By applying the inverse mask to the original frame, we clear the chessboard area. Then, using add operation, we overlay the transformed static image onto this cleared space.

The end result is a video frame where the static image appears naturally integrated into the scene, precisely overlaying the target area.

This process showcases a neat trick of computer vision, enabling us to blend digital content with the real world in a convincing way.

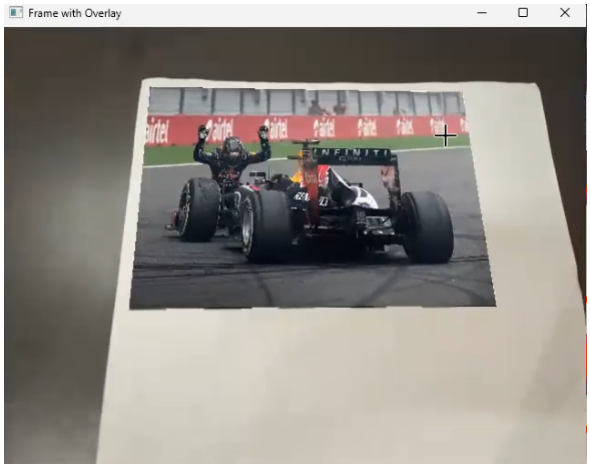

Fig. 13. Static image with target

4) *Add a virtual object, but also do something to the target to make it not look like a target any more*

In this task, we delved deeper into our endeavor to blend the virtual with the real, we harnessed sophisticated computer vision techniques to seamlessly overlay a virtual object on a detected target, effectively obscuring it.

1. Initial Detection: Utilizing grayscale conversion of the

video frame enhances the efficacy of subsequent processes, setting the stage for precise feature detection, in this case, a chessboard.

2. Perspective Transformation: The `'getPerspectiveTransform'` function crafts a mathematical blueprint to map the static image onto the chessboard's exact perspective. This transformation is pivotal for the subsequent `'warpPerspective'` step, which meticulously adjusts the static image to conform to the chessboard's perspective, ensuring a seamless integration.

3. Mask Creation and Application: Through `'fillConvexPoly'`, we generate a mask delineating the chessboard's bounds. This mask, complemented by its inverse obtained via `'bitwise_not'`, allows for strategic isolation and removal of the chessboard, paving the way for the virtual overlay's insertion without disrupting the frame's integrity.

4. Virtual Object Projection: Employing `'solvePnP'` yields the rotation and translation vectors critical for anchoring our virtual structure in three-dimensional space relative to the camera's viewpoint. This step is the linchpin for accurately projecting the virtual object onto the real-world scene.

5. 3D to 2D Mapping: The `projectPoints` function transposes the virtual object's 3D coordinates into the 2D plane of our video frame, informed by the camera's calibration parameters and the aforementioned vectors, ensuring the object's visual coherence within its newfound setting.

6. Drawing and Displaying: Finally, we dynamically render the virtual object atop the frame by connecting its projected 2D points using line, effectively breathing life into our virtual creation. The object not only masks the target but also adopts a convincingly tangible presence within the video stream.

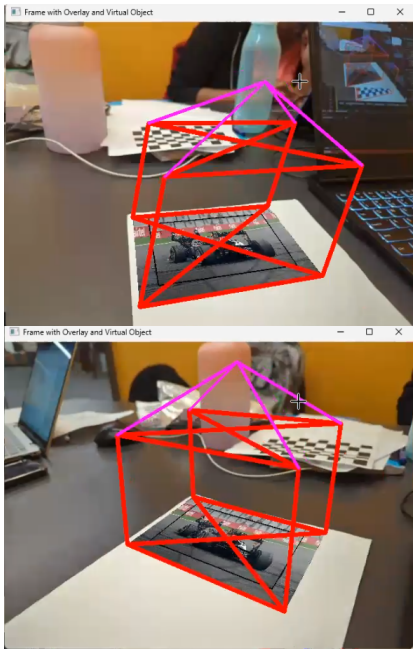

Fig. 14. Virtual object with target surface different

This intricate blend of detection, transformation, and projection techniques underscores our technical journey from merely overlaying static images to embedding complex virtual objects into live scenes, broadening the horizons for augmented reality applications.

- Live demo video-1 of working system
- Live demo video-2 of working system

II. What we learned!!!

- Spotting Details in Pictures: We got to grips with techniques like the Harris corner detection, which helps find specific points in images that are key for recognizing objects or features. It's like teaching a computer to spot the most interesting parts of a picture.
- Hands-on with Vision Tech: We rolled up our sleeves and used a toolkit called OpenCV in C++ to make our computers process live video. This meant making them smart enough to find these interesting points on their own as the video plays.
- Tuning the Camera's Eye: Cameras don't see the world perfectly, so we learned how to adjust them to see as clearly as possible. This involves fixing common issues like lens distortion, which can make straight lines look bent. It's a bit like putting on glasses to see better.
- Bringing Virtual Objects to Life: We discovered how to take objects that only exist in the computer and make them appear as if they're part of the real world, captured by the camera. This involves a lot of math to make sure they look like they're really sitting on your table or hanging out in your room.

III. Acknowledgement

Acknowledging the wealth of knowledge available online, particularly in the 'OpenCV' documentation and tutorials, played a crucial role. This project's success is a testament to the collective effort of the educational community, emphasizing the shared wealth of expertise. Interactions with ChatGPT proved invaluable for exploring concepts and troubleshooting, enhancing the overall learning experience of the project. This acknowledgment is a tribute to the collaborative and supportive environment that significantly facilitated the learning journey in this project.